Pascal Pons Follow

SOLVING CONNECT FOUR

Part 4 – Alpha-beta algorithm

The alpha-beta algorithm

Alpha-beta pruning leverages the fact that you do not always need to fully explore all possible game paths to compute the score of a position. For example if it’s your turn and you already know that you can have a score of at least 10 by playing a given move, there is no need to explore for score lower than 10 on other possible moves.

More generally alpha-beta introduces a score window [alpha;beta] within which you search the actual score of a position. It relaxes the constraint of computing the exact score whenever the actual score is not within the search windows:

- If the actual score of the position is within the range, than the alpha-beta function should return the exact score.
- If the actual score of the position lower than alpha, than the alpha-beta function is allowed to return any upper bound of the actual score that is lower or equal to alpha.
- If the actual score of the position greater than beta, than the alpha-beta function is allowed to return any lower bound of the actual score that is greater or equal to beta.

Relaxing these constrains allows to narrow the exploration window, taking into account other possible moves already explored. It also allows to prune the search tree as soon as we know that the score of the position is greater than beta.

At the beginning you should ask for a score within [-∞;+∞] range to get the exact score of a position.

Negamax implementation

We will keep implementing the negamax variant of alpha-beta. Here is the main function:

```
/**
 * Reccursively score connect 4 position using negamax variant of alpha-beta algorithm.
 * @param: alpha < beta, a score window within which we are evaluating the position.
 *
 * @return the exact score, an upper or lower bound score depending of the case:
 * - if actual score of position <= alpha then actual score <= return value <= alpha
 * - if actual score of position >= beta then beta <= return value <= actual score
 * - if alpha <= actual score <= beta then return value = actual score
 */
int negamax(const Position &P, int alpha, int beta) {
  if(P.nbMoves() == Position::WIDTH*Position::HEIGHT) // check for draw game
    return 0;

  for(int x = 0; x < Position::WIDTH; x++) // check if current player can win next move
    if(P.canPlay(x) && P.isWinningMove(x))
      return (Position::WIDTH*Position::HEIGHT+1 - P.nbMoves())/2;

  int max = (Position::WIDTH*Position::HEIGHT-1 - P.nbMoves())/2;       // upper bound of our score as we cannot win immediately
  if(beta > max) {
    beta = max;                     // there is no need to keep beta above our max possible score.
    if(alpha >= beta) return beta;  // prune the exploration if the [alpha;beta] window is empty.
  }

  for(int x = 0; x < Position::WIDTH; x++) // compute the score of all possible next move and keep the best one
    if(P.canPlay(x)) {
      Position P2(P);
      P2.play(x);               // It's opponent turn in P2 position after current player plays x column.
      int score = -negamax(P2, -beta, -alpha); // explore opponent's score within [-beta;-alpha] windows:
      // no need to have good precision for score better than beta (opponent's score worse than -beta)
      // no need to check for score worse than alpha (opponent's score worse better than -alpha)

      if(score >= beta) return score;  // prune the exploration if we find a possible move better than what we were looking for.
      if(score > alpha) alpha = score; // reduce the [alpha;beta] window for next exploration, as we only
      // need to search for a position that is better than the best so far.
    }
  return alpha;
}
```

Check the full source code corresponding to this part.

Weak solver

Another benefit of alpha-beta is that you can easily implement a weak solver that only tells you the win/draw/loss outcome of a position by calling evaluating a node with the [-1;1] score window.

Benchmark

The performance evaluation shows that alpha-beta pruning reduces significantly the number of explored node, allowing to solve more complex positions.

Solver	Test Set name	mean time	mean nb pos	K pos/s
Alpha-beta (strong solver)	End-Easy	69.62 μs	283.6	4,074
Alpha-beta (strong solver)	Middle-Easy	4.54 s	54,236,700	11,940

Solver	Test Set name	mean time	mean nb pos	K pos/s
Alpha-beta (strong solver)	Middle-Medium	38.7 s	453,614,000	11,725
Alpha-beta (strong solver)	Begin-Easy	N/A	N/A	N/A
Alpha-beta (strong solver)	Begin-Medium	N/A	N/A	N/A
Alpha-beta (strong solver)	Begin-Hard	N/A	N/A	N/A
Alpha-beta (weak solver)	End-Easy	52 μs	222.9	4,284
Alpha-beta (weak solver)	Middle-Easy	3.28 s	41,401,200	12,638
Alpha-beta (weak solver)	Middle-Medium	24.5 s	308,114,000	12,548
Alpha-beta (weak solver)	Begin-Easy	N/A	N/A	N/A
Alpha-beta (weak solver)	Begin-Medium	N/A	N/A	N/A
Alpha-beta (weak solver)	Begin-Hard	N/A	N/A	N/A

mean time: average computation time (per test case). *mean nb pos*: average number of explored nodes (per test case). *N/A* means that the algorithm was too slow to evaluate the 1,000 test cases within 24h.

Tutorial plan

SOLVING CONNECT FOUR

Updated: February 20, 2017

Previous Next